Java Script

Rev. 5.2 del 10/11/2025

Java Script

Con il termine **script** si intende un **insieme di comandi interpretati da un certo applicativo** che, nel caso di Java Script, è costituito dal browser. Java Script è un linguaggio che consente di scrivere, all' interno di una pagina HTML, delle sezioni di codice che verranno interpretate dal browser al momento della visualizzazione della pag

Un tag HTML può richiamare uno script js mediante l'utilizzo dei cosiddetti **ATTRIBUTI DI EVENTO** che, in risposta ad un certo evento come ad esempio un click, consentono di associare delle righe di codice da eseguire nel momento in cui si verificherà l'evento. La potenzialità di java script consiste principalmente nel fatto che, indipendentemente dal tag che ha scatenato l'evento, **le istruzioni possono accedere in modo diretto a qualunque tag della pagina e modificarne sia l'aspetto grafico sia il contenuto**.

Le istruzioni Java Script possono essere scritte:

- Direttamente all'interno del tag (se le azioni da compiere sono poche)
```
<input type="button" value = "Esegui" onClick="alert('salve')">
```
- All'interno della sezione di HEAD della pagina all'interno del seguente TAG
```
<script type="application/javascript" >
    istruzioni js
</script>
```
- In un file esterno con estensione .js richiamato nel modo seguente:
```
<script type="application/javascript" src="index.js"> </script>
```

Sintassi base

- Le **istruzioni base** sono le stesse dell'ANSI C e sono **case sensitive** (distinzione maiuscole / minuscole)
- Il **punto e virgola** finale è facoltativo. Diventa obbligatorio quando si scrivono più istruzioni sulla stessa riga.
- Lo switch accetta come parametro anche le stringhe (come in C#, a differenza del C ANSI)
- Java Script non distingue tra **virgolette doppie** e **virgolette semplici.** Ogni coppia deve essere dello stesso tipo
  Per eseguire un terzo livello di annidamento si può usare \" oppure `&quot;` : alert ("salve \"Mondo\" ");
- Le variabili dichiarate fuori dalle funzioni hanno visibilità globale ma **non vengono inizializzate.**

Dichiarazione delle variabili

```
let A, B;
```

- **Le variabili non sono tipizzate**, cioè nella dichiarazione non deve essere specificato il tipo. I tipi sono comunque **Number, String** (object), **string** (scalare), **Boolean, Object**, e il cast al tipo viene eseguito in automatico in corrispondenza di ogni assegnazione. Il tipo **Float** non esiste (usare Number) ma esiste parseFloat( ). Oltre a **`let`** è possibile utilizzare **`const`** per le costanti
- E' comunque possibile tipizzare una variabile creando una istanza esplicita:
```
let n=new Number();
let s=new String();
let b=new Boolean();    // true e false sono minuscoli.
```
  Le parentesi tonde sono quelle del costruttore e possono indifferentemente essere scritte oppure omesse.
- **La dichiarazione delle variabili non è obbligatoria**. Una variabile non dichiarata viene automaticamente creata in corrispondenza del suo primo utilizzo **come variabile globale**, e dunque sarà visibile anche fuori dalla funzione. Facile causa di errori. "**`use strict`**" rende obbligatoria la dichiarazione delle variabili
- Al momento della dichiarazione ogni variabile viene inizializzata al valore **`undefined`** che non equivale a \0 e nemmeno a false, ma equivale a "non inizializzato". Si può anche testare: **`if(A == undefined)`**
- Per i riferimenti si utilizza il valore **`null`**
- Il test **`if (!myVar)`** da esito positivo se myVar è uguale ad **undefined** o **null** o **false** o ""
- Un **json vuoto** { } è invece un oggetto, quindi diverso da tutti i valori precedenti
- In javascript non è consentito passare parametri scalari per riferimento (occorre trasformarli in object)

Funzioni Utente

Occorre omettere **sia** il tipo del parametro **sia** il tipo del valore restituito.

```
function eseguiCalcoli(n) {
     istruzioni;
     return risultato;  }
```

"use strict"

Inserito sulla prima riga di un file js obbliga l'utente a dichiarare le variabili.

L'accesso agli elementi della pagina

Il metodo `document.getElementById` consente di accedere al TAG avente l'ID indicato come parametro. Restituisce un riferimento all'elemento indicato. [In caso di più elementi con lo stesso ID ritorna il primo che trova].

```
let _div = document.getElementById("txtNumero");
let n = parseInt(_div.value);
```

Il metodo `getElementsByName`

Notare il plurale (**Elements**) necessario in quanto il **`name`** non è univoco e quindi gli elementi individuati sono normalmente più d uno. Restituisce un oggetto **NodeList** contenente tutti gli elementi aventi il **`name`** indicato, nell'ordine in cui sono posizionati all'interno della pagina. Un **NodeList** è in pratica un **vettore enumerativo** accessibile tramite indice numerico e con la proprietà .length.
Anche nel caso in cui sia presente un solo elemento viene comunque **sempre** restituito un vettore (lungo 1).

```
let _opts = document.getElementsByName("optHobbies");
  for (let i=0;i< _opts.length;i++)
    console.log(_opts[i].value);
```

I meotdi `getElementsByTagName` e `getElementsByClassName`

La prima restituisce sotto forma di **NodeList** (vettore enumerativo) tutti i tag di un certo tipo (es DIV).
La seconda restituisce sotto forma di **NodeList** (vettore enumerativo) tutti i tag che implementano una certa classe.
Entrambe accedono a figli e nipoti. Restituiscono un **NodeList** anche nel caso in cui sia presente un solo elemento:

```
let _body= document.getElementsByTagName("body")[0];
_body.style.backgroundColor="blue";
let _carte = document.getElementsByClassName("carta");     // senza il puntino
_carte[i].style.backgroundColor="red";
```

Ricerche annidate

```
let _wrapper = document.getElementById("wrapper")
let vet = _wrapper.getElementsByTagName("p");  // tag <p> interni a wrapper
```

Nota: **`getElementById`** e **`getElementsByName`** possono essere applicati SOLO a **document** .

Accesso agli attributi HTML e alle proprietà di stile

Dopo aver 'agganciato' il puntatore ad un elemento della pagina:

- Si può accedere a tutti i suoi **attributi HTML** mediante l'uso dell'operatore puntino : txtRis.value = 72
  Gli attributi HTML sono sempre di tipo STRING. La conversione da numero a string è fatta in automatico
- Si può accedere a tutte le sue **proprietà CSS** mediante l'uso dell'attributo **`.style`**
  `div.style.border = "2px solid black";`
  La proprietà CSS contenenti un trattino (come ad esempio **`border-color`**) in javascript devono essere scritte con tutto attaccato in camelCasing: ad esempio **`borderColor`**

Le proprietà .innerHTML e .textContent

Per accedere al contenuto interno di un tag (quello scritto tra **<apertoTag>** e **</chiusoTag>** si possono utilizzare due attributi dinamici (nel senso che esistono solo in javascript e non nella pagina html) che sono `.innerHTML e .textContent.` La differenza consiste nel fatto che:

innerHTML Accetta come valore qualsiasi tag HTML e restituisce l'interno contenuto html presente nel tag
textContent accede/imposta il solo contenuto testuale del tag (anche se annidato al''interno di più sotto-tag).

```
myDiv.innerHTML = "<i>Questo è il mio testo</i>"
```

Sono disponibili anche altri due attributi "minori" :

outerHTML mentre innerHTML accede al contenuto del tag (tag radice escluso), outerHTML restituisce anche il tag radice (cioè il tag al quale si applica la property). Può essere usata per cambiare anche il tag radice.

innerText simile a textContent, standardizzata più tardi nel DOM model. Leggermente più pesante. Differenze:

- Restituisce anche eventuali attributi hidden
- Riconosce il \n
- E' definita solo nel caso degli HTMLElement (che sono un sottoinsieme dei NodeElement definiti in XML)

L'evento onload

Nel file JS è possibile scrivere ***codice diretto*** "esterno" a qualsiasi funzione. Questo codice viene però eseguito **prima** che sia terminato il caricamento della pagina per cui **non** può fare riferimento agli oggetti della pagina che verranno istanziati soltanto al termine del caricamento. In tal caso i riferimenti saranno tutti NULL.

Per eventuali inizializzazioni occorre utilizzare, all'interno del file html, l'evento **<body onload="init()">**

Anziché definirlo all'interno della pagina HTML, l'evento **onLoad** può essere definito direttamente all'interno del file javascript come evento dell'oggetto window (finestra corrente)

```
window.onload = function(){ ..... }      // window può essere omesso
```

L'evento viene però generato ***SOLO SE*** non è già stato definito all'interno del body html (che è prioritario).

Gli oggetti base

L'oggetto Math

Proprietà Statiche

Math.PI	PI greco	3.1416
Math.E	Base logaritmi naturali	2.718

Math.LN2	Logaritmo naturale di 2
Math.LN10	Logaritmo naturale di 10
Math.LOG2E	Logaritmo in base 2 di e
Math.LOG10E	Logaritmo in base 10 di e

Metodi Statici

Math.sqrt(N)	Radice Quadrata
Math.abs(N)	Valore assoluto di N (modulo)
Math.max(N1, N2)	Math.min(N1,N2) Restituiscono il maggiore / minore in una sequenza di n numeri
Math.pow(10,N)	Elevamento a potenza : 10^N // oppure 10 ** N
Math.round(N)	Arrotondamento all'intero più vicino
Math.ceil(N)	Arrotondamento all'intero superiore
Math.floor(N)	Tronca all'intero inferiore. **L'operatore di divisione / restituisce un double anche nel caso di divisione fra interi, per cui il risultato deve eventualmente essere troncato.**
%	**restituisce il resto di una divisione fra interi**
Math.random()	Restituisce un double 0 <= x < 1 **esattamente come in ANSI C.** *Randomize è automatico* Per generare un num tra A e B (A compreso, B escluso) `Math.floor((B-A)*Math.random())+ A`

Metodi di istanza: toFixed()

```
let n = 12.34567;
alert(n.toFixed(2))     // 12.34
```

Come separatore delle migliaia sui numeri si può usare l'**underscore** (ES2021) che non influisce sul valore numerico, ma solo per migliorare la leggibilità.

Le Stringhe

String è un oggetto che deve pertanto essere istanziato**.** Le stringhe sono **immutabili** come in C#
Per dichiarare una stringa sono disponibili le due seguenti sintassi (l'istanza statica **`String s1`** non è supportata):

```
let s1;                    // riferimento non tipizzato. Contiene undefined
let s2 = new String();  // riferimento tipizzato ma che contiene sempre undefined
```

Inizializzazione di una stringa in fase di dichiarazione

```
let s3 = "salve";                 // oggetto stringa inizializzato a "salve"
let s4 = new String("salve");  // oggetto stringa inizializzato a "salve"
let len = s1.length    // lunghezza della stringa
```

s1 = s1.**toUpperCase**() // Restituisce la stringa convertita in maiscuolo
s2 = s1.**toLowerCase**() // Restituisce la stringa convertita in minuscolo
s2 = s1.**substr**(posIniziale, [qta]) // Estrae i caratteri da posIniziale per una lunghezza pari a qta. Deprecato
s2 = s1.**substring**(posIniziale, posFinale) // Estrae i caratteri da posIniziale a posFinale, posFinale escluso.
s1 = "Salve a tutti" s2=s1.substring(0, 5) => "Salve". Se posFinale > length si ferma a fine stringa
ris = s1.**includes**(s2) // Restituisce **true** se s1 include s2. **false** in caso contrario. **Disponibile anche sui vettori**
pos = s1.**indexOf**(s2, [pos]) // Ricerca s2 dentro s1 a partire dalla posizione pos. (Il primo carattere ha indice 0).
Restituisce la posizione della prima ricorrenza. Se non ci sono occorrenze restituisce –1
Se s2 = "" restituisce il carattere alla posizione pos. **Disponibile anche sui vettori**
pos = s1.**lastIndexOf**(s2, [pos]) // Ricerca s2 dentro s1 a partire dalla posizione pos andando all'indietro.
Se pos non è specificato parte dalla fine della stringa (length – 1).
pos = s1.**search**(s2) // Identica a indexOf ma **accetta le Regular Expr. N**on è disponibile sui vettori
s2 = s1.**replace**("x", "y") // Sostituisce **SOLO la prima occorrenza**.
replaceAll("x", "y") // Sostituisce tutte le occorrenze
s2 = s1.**repeat**(3) // Ripete 3 volte il contenuto di s1
ris = s1.**startsWith**(s2) // Restituisce TRUE se s1 inizia con la sottostringa s2
ris = s1.**endsWith**(s2) // Restituisce TRUE se s1 termina con la sottostringa s2

C = s1.**charAt**(pos) // Restituisce come **stringa** il carattere alla posizione pos (partendo a 0)
N = s1.**charCodeAt**(pos) // Restituisce il **codice Ascii** del carattere alla posizione pos (o del primo se manca pos)
s2 = **String.fromCharCode**(97,98,99) Viene istanziata una nuova stringa contenente "abc"
s2 = n.**toString**() // Restituisce la conversione in stringa. **n.toString(16)** restituisce una stringa esadecimale
n.toString(2) restituisce una stringa binaria
vect = s.**split**("separatore"); // Esegue lo split rispetto al carattere indicato.
s = vect.**join**("separatore"); // Restituisce in un'unica str tutti gli elementi del vett separati dal chr indicato

n.**toFixed**(3) // indica il numero di cifre dopo la virgola da visualizzare

A differenza di Ansi C in js le stringhe possono essere **confrontate** direttamente con gli operatori < >

Una semplice funzione per aggiungere uno 0 davanti ad un numero <10

```
function pad(number) {
     return (number < 10 ? '0' : '') + number
}
```

Altri metodi relativi alla formattazione grafica

.big()	restituisce una stringa in testo grande
.blink()	restituisce una stringa con testo lampeggiante
.bold()	restituisce una stringa in grassetto
.fontsize()	restituisce una stringa avente il fontsize specificato
.italics ()	restituisce una stringa in corsivo
.small()	restituisce una stringa in testo piccolo
.strike()	restituisce una stringa barrata
.sup()	restituisce una stringa in formato apice .sub() pedice

Vettori Enumerativi (Indexed Array cioè vettori indicizzati)

I vettori sono liste a indice 0 e possono essere dichiarati anche SENZA specificare la dimensione.
Il vettore crescerà dinamicamente man mano che si aggiungeranno elementi al suo interno.
Per creare un vettore sono disponibili due sintassi equivalenti, una breve ed una più prolissa:

let vect = [ ]; // dichiaro un vettore enumerativo al momento avente lunghezza 0
let vect = **new Array**() // Simile alla precedente ma tipizzata

L'assegnazione della dimensione in fase dichiarativa può essere realizzata soltanto mediante la seconda sintassi:
let vect = **new Array**(30) // Viene istanziato un array di 30 elementi

Attenzione che invece la seguente sintassi dichiara un vettore lungo 1 contenente il valore 30:
let vect = [30]; // Viene istanziato un array contenente un solo elemento, con valore 30.

Inizializzazione di un vettore in fase di dichiarazione

let vect = **new Array**(30,31) // Viene istanziato un array di **2** elementi, contenente i valori 30 e 31
let vect = [30, 31]; // uguale

let vect = **new Array (**'pippo'**);** // Viene istanziato un array di **1** elemento, contenente 'pippo'
let vect = [titolo, autore, categoria, prezzo]; // Viene caricato nel vettore il contenuto delle variabili indicate

Proprietà e Metodi di un vettore enumerativo

Gli Array possono essere **eterogenei**, cioè contenere dati differenti: numeri, stringhe, oggetti, etc
Per accedere all'i-esimo elemento si usano le **parentesi quadre** vectStudenti[3] = "Mario Rossi"

```
vect.length          // lunghezza del vettore
vect.push("a", "b", "c");   // Gli elementi indicati vengono aggiunti in coda al vettore
                            // Accetta come parametro anche un intero vettore che viene concatenato
vect.unshift("a", "b", "c");      // Gli elementi indicati vengono aggiunti in testa al vettore
vect.pop( );         // Restituisce l'ultimo elemento in coda al vettore, eliminandolo dal vettore
vect.shift( );       // Restituisce il primo elemento in testa al vettore, eliminandolo dal vettore
vect[vect.length] = "value";   // Equivalente al push()
vect[99] = "value";     // Se il vettore fosse lungo 10, verrebbe create 90 celle,
                           con le celle 10-98 undefined, ma comunque con length==100
pos = vect.indexOf(item)      // Restituisce la posizione dell'elemento indicato.
ris = vect.includes(item)     // Restituisce true se vect include l'elemento indicato.
                           // Per quanto riguarda gli oggetti si rimanda al modulo "XML and JSON"
```

`vect.splice(pos, n);` // Consente di eliminare n oggetti a partire dalla posizione pos.
Restituisce un **vettore** con gli elementi eliminati, anche se l'elemento è uno solo.
`vect.splice(-1, 2);` elimina due elementi all'indietro partendo dall'ultimo.
`delete vect[3]` cancella il contenuto della cella 3, **senza però ricompattare il vettore** !!

Eventuali parametri successivi consentono di aggiungere nuovi elementi alla posizione indicata.
`vect.splice(pos,3,"A","B")` Rimuove 3 elementi dalla posizione pos e, al loro posto, inserisce A e B
Passando `0` come secondo parametro, "A" e "B" vengono inseriti nella posizione **`pos`** senza cancellare nulla.
`vect.splice(pos,0,"A","B")` inserisce A e B nella posizione **`pos`** senza cancellare nulla

```
vect.sort();          // Ordinamento crescente. Agisce direttamente sul vettore, senza doverlo riassegnare
vect.reverse();       // Ordinamento decrescente.    // Il vettore viene comunque anche restituito
```

E possibile passare a sort una funzione di confronto fra celle da implementare manualmente a seconda delle esigenze

```
students.sort(function(item1, item2) {
        if (item1 > item2)    return 1;    // inverte
        else        return -1;             // non inverte
});
```

Il metodo sort scorre il vettore per ordinarlo e, in corrispondenza di ogni confronto, richiama la funzione ricevuta come parametro iniettandogli i due valori **item1** e **item2**. **Se la funzione ritorna un qualunque numero postivo il sort esegue l'inversione fra le due celle, altrimenti non inverte**.

Questa funzione può essere usata in modo efficace per 'mescolare' casualmente il contenuto di un vettore :

```
vet.sort(function (a, b) {return Math.random() - 0.5);
```

Math.random() restituisce un numero decimale tra 0 e 1. Se il numero è > 0.5 inverte, altrimenti NON inverte.

let vect2= vect.**slice**(inizio, fine); // Restituisce un nuovo vettore contenente gli elementi indicati da ***inizio*** (compreso) a ***fine*** (escluso). Numeri negativi consentono di selezionare a partire dal fondo.
let vect2= vect.**slice**(2, -2); // Restituisce dalla cella 2 fino alla penultima (esclusa). **Disponibile anche sulle stringhe**

La scansione di un vettore enumerativo

```
for(let item of vet) alert(item);      // item rappresenta il contenuto della cella
```

Attenzione che la variabile di ciclo **item** (detta **cursore**) è una **copia** del contenuto della cella, per cui eventuali modifiche apportate al cursore vanno perse al termine del ciclo.

Il metodo funzionale **vet.forEach(function(item, i))** consente di scorrere un vettore enumerativo analogamente al for-of, ma essendo asincrono è più veloce. Il 2° parametro i è facoltativo

Inserimento di una variabile all'interno di una stringa

Il **backtick** (apice singolo rovesciato = **ALT 96)** è uno speciale delimitatore di stringa che:

- consente di inserire delle variabili direttamente all'interno della stringa tramite **${varName}**
- consente di **andare a capo** all'interno della stringa, che può essere scritta su più righe di testo

```
`Sono l'elemento ${i} Il mio valore è ${vet[i]}`
```

Le Matrici

In Java Script non esiste il concetto canonico di matrice. Una matrice è semplicemente un vettore di vettori.
Esempio di creazione di una **matrice 10 x 3** :

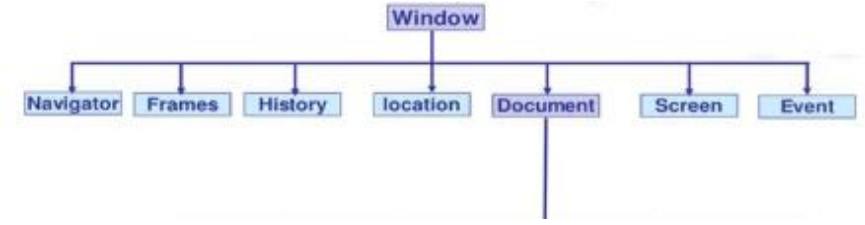
```
let mat = new Array (10);
for (i=0; i < mat.length; i++)
    mat[i] = []       // oppure mat[i] = new Array (3);

mat[0][0] = "Marsha";   mat[0][1] = "Carol";   mat[0][2] = "Sandy"
mat.length      //  restituisce la lunghezza del vettore principale, cioè 10
mat[0].length   //  restituisce la lunghezza del primo vettore interno, cioè 3
```

Il DOM di una pagina HTML

Quando una pagina HTML viene caricata, il browser crea un DOM (Document Object Model) della pagina, cioè una rappresentazione JavaScript degli elementi HTML mettendo a disposizione un insieme di oggetti e funzioni che consentono di accedere a tutti gli elementi della pagina HTML.

L'oggetto base del DOM è l'oggetto **window** che rappresenta la scheda di navigazione corrente.



I principali oggetti figli dell'oggetto windows sono:

- **document** = documento html caricato nella finestra (analogo di xmlDoc nei documenti XML)
- **location** = informazioni sulla url corrente
- **navigator** = oggetto di navigazione. Utile ad esempio per eseguire dei redirect.

Principali Metodi dell'oggetto window

window è l'oggetto predefinito del DOM, per cui i suoi metodi sono utilizzabili anche omettendo window che può SEMPRE essere omesso. **I metodi iniziano con un verbo. Quelli che non iniziano con un verbo sono property**

alert("Messaggio") Visualizza una finestra di messaggio con un unico pulsante OK non modificabile.
prompt("Messaggio") Rappresenta il tipico Input Box con i pulsanti OK e ANNULLA. Un secondo parametro opzionale indica un valore iniziale da assegnare al campo di immissione. Clickando su OK restituisce come stringa il valore inserito. Clickando su ANNULLA restituisce **null**
confirm("Messaggio") Finestra di conferma contenente i due pulsanti OK e ANNULLA. Restituisce **true** se l'utente clicka su OK oppure **false** se l'utente clicka su ANNULLA. Il test sul boolean è in genere diretto: if (confirm("Vuoi veramente chiudere?")) { …….. }

NaN Not a Number. Quando una operazione matematica non è valida (Math.sqrt(-1)) js restituisce **NaN** che non significa che il risultatno NON è un numero, ma significa che è un numero non valido ! Ad esempio `typeof NaN == "number"` VERO !!
isNaN(s) Consente di testare se la variabile s contiene il valore NaN, nel qual caso restituisce true.
parseInt(s) Converte una stringa o un float in numero intero.
In caso di stringa **si ferma automaticamente al primo carattere non numerico**.
In caso di float **il numero viene troncato all'intero inferiore** (come `Math.floor()`)
nota: .value di un input type=number, a differenza di .innerHTML, nel caso di contenuti numerici restituisce non una stringa ma un numero intero.
parseFloat(s) Converte una stringa in float
Number(s) E' simile a ***parseFloat*** però esegue la conversione SOLO se s contiene SOLTANTO caratteri numerici. Se c'è anche una sola lettera (15A) restituisce NaN. Accetta la virgola.
Se c'è un carattere alfanumerico restituisce NaN. In caso di " " restituisce 0
toString( ); Converte qualsiasi variabile in stringa

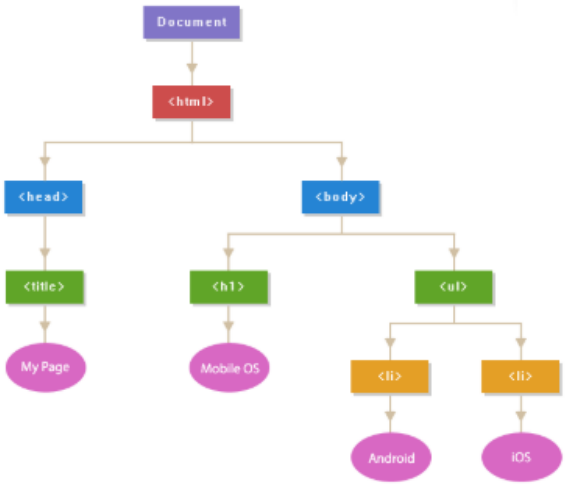
open `("file.html", [target], ["Opzioni separate da virgola"])`
target: `"_self"` apre la nuova pagina nella stessa scheda
`"_blank"` apre la nuova pagina in una nuova scheda
A differenza di `<a href>` in cui il default di target era `"_self"`, questa volta il default è `"_blank"`
Come target si può passare anche il nome di un `iframe`.
close( ) Chiude la finestra corrente se questa è stata aperta da codice tramite il metodo .open().
Se la finestra se è stata aperta in locale funziona anche in caso di apertura da browser, se invece proviene da un server NO..
scroll(x,y) Fa scorrere la finestra di x colonne orizzontalmente e y righe verticalmente.
focus() / blur() Porta la finestra in primo piano / sotto le altre finestre
self window.self **è** un puntatore alla window medesima (come se fosse un alias)

encodeURIComponent(s) Codifica gli spazi e tutti i caratteri speciali nei rispettivi codici **utf-8** in formato esadecimale. Ad esempio lo spazio viene codificato come %20 :
escape ("±salve mondo") diventa **"%B1**salve**%20**mondo".
decodeURIComponent(s) decodifica i caratteri utf-8 restituendo la stringa originale.
escape(s) / unescape(s) analoghe alle precedenti ma deprecate

Metodi di evento
onLoad() Richiamato in corrispondenza del caricamento del documento
onUnload() Richiamato quando si abbandona un documento (sostituito da un nuovo documento)
onResize() Richiamato quando la finestra attiva viene ridimensionata.

L'oggetto *document* e la gestione degli elementi della pagina

Rappresenta il documento HTML che si sta visualizzando all'interno dell'oggetto window.
Sono figli di **`document`** tutti i tag della pagina html, rappresentabili mediante la seguente struttura **ad albero**:



I cerchi fucsia rappresentano le foglie dell'albero

I controlli

Proprietà comuni (che valgon SOLO per i controlli)

disabled true / false. In caso di **disabled=true** il controllo assume un aspetto grigio e non risponde più agli eventi. Vale **SOLO** nel caso dei controlli e **per il tag BUTTON.**

value esiste **SOLO** nel caso dei controlli e rappresenta il contenuto del tag medesimo.
Il tag ***button*** non dispone della proprietà ***value*** ma occorre utilizzare ***innerHTML*** o ***textContent***

Metodi ed Eventi comuni

onclick() usato principalmente per i **`pulsanti`**
onchange() usato per i tag input e generato dopo che è cambiato il contenuto del campo, ma soltanto quando si rilascia il mouse. Usato principalmente per **`radiobuttons`** **`checkbox`** e **`listbox`**
focus() metodo che assegna il fuoco all'oggetto
onfocus() / onblur() eventi generato dopo che l'oggetto ha ricevuto/perso il fuoco

input[type=button]

Scopo di questo pulsante (a differenza dell'input **type=submit**) è quello di richiamare una procedura javascript.
Per gli <input[type=button]> si usa il **`value`**, mentre per i <button> si usa **`textContent/innerHTML`**
Il metodo **`.click()`** consente di forzare un click sul pulsante

Text Box

value rappresenta il contenuto del campo, anche nel caso delle TEXT AREA.
oninput() evento generato alla pressione di ogni tasto. Richiamato ***DOPO*** che il contenuto del TextBox è stato aggiornato ma ***PRIMA*** che venga visualizzato E' l'evento migliore ad es per convertire in maiuscolo ma *non vede il keyCode del tasto premuto.* Usato per **`textbox`** **`textarea`** **`e`** **`slider`** <range>
onchange() evento generato dopo che è stata apportata una modifica rispetto al valore iniziale (quando ha ricevuto il focus), ma generato SOLO nel momento in cui si preme il tasto ***ENTER,*** oppure quando il textbox perde il focus
select( ) metodo che seleziona il contenuto del textbox
onSelect() generato dopo che il contenuto del campo è stato (anche solo parzialmente) selezionato.

TextArea

Come visto negli appunti di HTML staticamente all'interno di una **textarea** è possibile inserire testo soltanto all'interno del tag e non all'interno dell'attributo **value** (che invece rappresenta il corrispondente **currentState** dinamico utilizzabile **SOLO** da javascript).

Ciò che verrà inserito dinamicamente dall'utente all'interno dell'interfaccia grafica viene dunque salvato nel **currentState** (***cioè nel value***). Se il currentState (value) è vuoto, value restituisce il valore il defaultState (scritto all'interno del tag nella pagina HTML). In javascript:

- la property **`.innerHTML`** restituisce sempre solo il **defaultState** statico
- la property **`.value`** restituisce il **currentState** dinamico che maschera completamente il precedente.

Quindi in js, nel caso della `textarea`, occorre utilizzare SEMPRE SOLO value
Anche in corrispondenza di un submit, ciò che viene trasmesso al server è sempre il value.

Radio Button

Sono mutualmente esclusivi soltanto gli option button con lo **stesso name**, che possono pertanto essere gestiti come un vettore di controlli.

.length	numero di pulsanti appartenenti al gruppo
[i].checked	true / false indica se il radio button è selezionato.
[i].value	Rappresenta il contenuto dell'attributo value (valore restituito in corrispondenza del submit se il radio button è selezionato)
onchange()	Deve essere associato ad ogni singolo radio button e viene generato ***soltanto*** in corrispondenza della selezione del radio button (cioè NON viene attivato quando il radio button viene deselezionato) e ***soltanto dopo che il cambio di stato è avvenuto***
onclick()	Uguale identico al precedente, però viene generato ad ogni click sul radio button, anche se il radio button è già selezionato.
[i].click()	Forza un click sul pulsante che cambia pertanto di stato. Notare che la selezione di un elemento da codice **NON provoca la generazione degli eventi onchange e onclick**, che devono essere esplicitamente richiamati da codice.

In java script NON è possibile associare gli eventi onClick e onChange ad una intera collezione di radio button (come in jQuery) ma occorre assegnare la procedura di evento per ogni singolo pulsante

Per vedere qual è la voce selezionata all'interno di un gruppo di radio buttons occorre pertanto fare un **ciclo** su tutti gli elementi del gruppo e valutare se i singoli attributi **checked** sono true/false
Oppure si può usare **`querySelector`**

```
let chk = document.querySelector("input[type=radio]:checked")
alert(chk.value)
```

CheckBox

Presentano un funzionamento simile rispetto ai Radio Button, con l'unica differenza che non sono esclusivi.
L'evento **onchange()** questa volta viene generato ad ogni cambio di stato, cioè su ogni singolo elemento, sia in selezione che in de-selezione

List Box <select>

Non riconosce placeholder

onchange()	Evento standard richiamato in corrispondenza della selezione di una nuova opzione. Non viene richiamato nel caso in cui da codice si imposti **`selectedIndex=-1`**
onClick	Evento richiamato ad ogni singolo click sulla Lista. Intercetta anche il click di apertura della lista.

Options

_myList.options restituisce una Collection contenente tutte le **options**.

.options.length	Numero di opzioni
.options[i].textContent	Testo visualizzato all'interno dell'opzione ("Fossano")
.options[i].value	Valore Nascosto interno all'opzione ('1')
.options[i].selected	true / false a seconda che l'opzione i-esima sia selezionata o meno
.options[i].onclick()	Non standard. **Firefox** lo riconosce sempre **Chrome** (febbraio 2024) lo riconosce *SOLTANTO* se l'attributo SIZE è > 0

Se un tag <option> non ha l'attributo value, allora il suo value viene automaticamente impostato al contenuto interno (textContent), però non come attributo statico HTML ma come proprietà del DOM.

Principali proprietà dell'oggetto <*SELECT*>

.length numero di `options` presenti nella lista. Equivale a **options.length**

.selectedIndex indice della voce selezionata. Ricordando che size rappresenta il n. di voci visualizzate nella lista

- Se size==0 (default, funzionamento **combo**) selectedIndex viene impostato a 0, cioè sulla prima voce selezionata. Per cui l'evento change sulla prima voce **NON** si verificherà mai. Impostando da codice selectedIndex=-1 viene visualizzata una riga iniziale vuota che scomparirà in corrispondenza della 1 selez.
- Se invece size>0 (lista tradizione aperta) selectedIndex viene automaticamente impostato a -1.

.value L'oggetto <Select> dispone di un comodissimo attributo riassuntivo **value** che, **in corrispondenza della selezione di una voce, contiene il value della <options> attualmente selezionata.**
Notare invece che **option button e check box**, essendo costituiti da un vettore di oggetti **privo di un contenitore esterno** come <select>, non dispongono di una proprietà riassuntiva **value** riferita all'intero vettore, per cui per vedere quale option è selezionato occorre fare un **ciclo** oppure utilizzare :checked.

.selectedOptions Collection delle sole ***options*** selezionate (ricordare che in presenza del'attributo HTML ***multiple*** potrebbero esserci più options selezionate). Strutturalmente è una collection analoga a ***options*** e presenta gli stessi campi. Se c'è una sola opzione selezionata sarà un vettore lungo 1.

Deselezione della prima voce

Al termine del caricamento la prima voce risulta automaticamente selezionata. Per deselezionarla si può assegnare :

```
_myList.selectedIndex=-1      // oppure
_myList.value=""
```

ma solo DOPO che la lista è stata riempita

sintassi di accesso alla voce selezionata

```
_myList.value                                          // value
_myList.options[_myList.selectedIndex].value           // value
_myList.options[_myList.selectedIndex].textContent     // contenuto testuale
_myList.selectedOptions[0].value                       // value
_myList.selectedOptions[0].textContent                 // contenuto testuale
```

Aggiunta di nuove Opzioni:

```
_myList.innerHTML  +=  "<option value='1'> Fossano </option>"     // oppure
_myList.add(new Option('Fossano', '1'))   // il secondo parametro è facoltativo
_myList.options[i] = new Option('Fossano', '1')
```

Il push NON è ammesso

```
_myList.remove(index)     // rimuove l'elemento indicato
```

Eventi relativi alla tastiera

onkeydown( ) // richiamato al momento della pressione del tasto **PRIMA** che il carattere arrivi al textbox
onkeyup( ) // richiamato al momento del ritorno su del tasto **DOPO** che il carattere è arrivato al textbox
// Entrambi riconoscono tutti i tasti speciali, compreso Enter

onkeydown consente di abbattere il carattere (cioè fare in modo che non 'arrivi' al textbox)
onkeyup viene richiamato ***DOPO*** che il textbox è stato aggiornato e quindi **NON** consente di abbattere il carattere
onkeypress**( )** // DEPRECATO. Viene comunque richiamato PRIMA che il carattere arrivi al textbox
Tra i tasti speciali riconosce solo **Enter** In tutti gli altri casi l'evento non viene richiamato.

L'oggetto event

A tutte le procedure di evento javascript viene **iniettato** un oggetto **`event`** che contiene diverse informazioni sull'evento stesso e sull'oggetto che lo ha scatenato**.**

- Nel caso dell'onclick eseguito nell'html, il parametro **`event`** **NON** viene iniettato automaticamente e, se lo si vuole utilizzare, occorre scriverlo esplicitamente sia in fase di chiamata sia come parametro della function
  `<button onclick="esegui(event)"> esegui </button>`
- Nel caso di associazione dinamica degli eventi (tramite onclick e addEventListener), il parametro **`event`**, in fase di chiamata, viene iniettato **automaticamente** alla procedura di evento.

La procedura di evento, se vuole utilizzare il parametro **`event`**, lo deve sempre **dichiarare esplicitamente**:

```
function esegui(event) { console.log(event.target.value) }
btn.addEventListener("click", function (event) { ..... })
```

Principali Proprietà del parametro event

`event.target` puntatore all'elemento che ha scatenato l'evento (derivazione mozilla) esattamente come il **`this`** e risulta comodo laddove il **`this`** non è utilizzabile (ad es Angular)
`event.key` carattere premuto: ad esempio `"a"`, `"A"`, **`"Enter"`**`(ascii 13)`

`event.currentTarget` puntatore all'elemento a cui è associato l'evento
`event.srcElement` puntatore all'elemento che ha scatenato l'evento (derivazione chrome)
`event.keyCode` Deprecato. Restituiva il codice ASCII del tasto premuto. Attenzione che nel caso delle lettere restituiva sempre il codice ascii della lettera MAIUSCOLA, anche se la lettera inserita è minuscola !
`event.type` contiene il nome dell'evento (es "click")
`event.timeStamp` Tempo trascorso (in millisecondi) dal caricamento della pagina
`event.clientX` coordinate X del mouse rispetto alla window corrente
`event.clientY` coordinate Y del mouse rispetto alla window corrente
`event.which` nel caso della tastiera e mouse contiene informazioni aggiuntive sul tasto premuto

Codifica dei Caratteri speciali

`keydown` e **`keyup`**, a differenza di **`keypress`** (che fra i tasti speciali intercetta SOLO **`"Enter"`**) consentono di intercettare anche tutti gli altri caratteri speciali che devono essere scritti come stringa.

```
"ArrowUp"    "ArrowDown"    "ArrowLeft"    "ArrowRight",    "Enter"
"PageUp"     "PageDown"
"Home"       "End"          "Delete"       "Backspace"
"Shift"      "Tab"          "Alt"          "AltGraph"       "Control"
"CapsLock"   "NumLock"
```

Come 'accettare" come validi solo alcuni caratteri

Soluzione1 (vecchio stile, richiamabile solo dall'html)

```
onkeydown="return controlla(event)"
function controlla(event){
    let ascii = event.keyCode
    if(ascii>=48 && ascii<=57) return true
    else return false
}
```

Soluzione 2 (richiamabile sia dall'html sia da javascript)

```
onkeydown="controlla(event)"

function controlla(event){
    let char = event.key
    if( ! (char >= '0' && char <= '9') )  // lascia passare solo i numeri
         event.preventDefault();
}
```

Il metodo `.preventDefault()` abbatte il carattere premuto che **NON** verrà visualizzato all'interno del textbox.
Entrambe le soluzioni funzionano SOLO con `onkeydown` *(e onkeypress)*

Come 'modificare' un carattere digitato da tastiera

La proprietà event.key è di sola lettura, quindi non può essere modificata. Per modificare un carattere si può utilizzare

- ***onkeydown*** per abbattere il carattere **prima** che 'arrivi' al textbox e poi concatenare a mano il nuovo chr

```
if(event.key =='x'){
   event.preventDefault();
  _txt.value += "y"   // Attenzione che questo += consente di eccedere maxlength
}
```

- ***onkeyup*** andando poi a modificare il contenuto del textbox **dopo** che il carattere è 'arrivato' al textbox

```
txt.value=Parola.value.toUpperCase();
```

Come 'intercettare' combinazioni di tasti

```
if (event.ctrlKey && event.key.toLowerCase() == 'z')    // CTRL+Z
```

Definizione degli eventi sul body

Gli eventi **keydown e keyup** possono anche essere definiti direttamente su body, nel qual caso l'evento viene sempre intercettato qualunque sia l'elemento della pagina su cui si è verificato:

```
<body onkeyup="elabora(event)" >
function elabora(event){
   switch(event.key){
      case "ArrowUp": spostaSu(); break;
      case "ArrowDown": spostaGiu(); break;
```

Eventi relativi al mouse

onmousedown( ) // Inizio click
onmouseup( ) // Fine click
onmouseover( ) // Ingresso del mouse sull'elemento
onmouseout( ) // Uscita del mouse dall'elemento
onmousemove( ) // Spostamento del mouse all'interno dell'elemento

Creazione di nuovi attributi personalizzati (nascosti)

Su tutti i tag HTML è possibile creare, sia staticamente nel file .html sia dinamicamente tramite javascript, dei nuovi attributi '***personali***' nascosti in cui salvare delle informazioni che potrebbero poi essere utili in corso d'opera. Notare che questi attributi **NON** vengono visualizzati dall' inspector.

```
_div.nuovoAttributo = "valore";
```

Aggiunta / Rimozione di una classe

```
_div.classList.add("className1", "className2")
_div.classList.remove("className")
_div.classList.toggle("className")            // se non c'è la aggiunge, altrimenti la toglie
_div.classList.replace("class1", "class2")    //  sostituisce class1 con class2
```

Per vedere se un certo elemento contiene una classe oppure no si usa il seguente metodo:

```
if (_div.classList.contains("className ")) ……...
```

Accesso diretto al DOM tramite querySelector()

Il metodo **`myDiv.children`** restituisce un vettore enumerativo contenente tutti i **figli diretti** del tag DIV corrente, nell'ordine in cui sono scritti nel file html, Es: **`myFirstDiv = myWrapper.children[0]`**

I metodi **`document.querySelector()`** e **`document.querySelectorAll()`** accettano come parametro qualunque selettore / pseudoselettore CSS **così come potrebbe essere scritto all'interno di un file CSS**.

- Il metodo **querySelector()** ritorna il primo elemento **discendente** (cioè compresi figli e nipoti) che corrisponde al selettore specificato
- Il metodo **querySelectorAll()** ritorna un **vettore enumerativo** di tutti i nodi corrispondenti al selettore.

document.querySelector()

Se si specifica un **ID** diventa sostanzialmente equivalente a `document.getElementById`.
Però attenzione che, a livello sintattico, a differenza di `document.getElementById`, nel caso di `querySelector` occorre anteporre un #, in quanto querySelector si aspetta un selettore CSS

```
let _wrapper = document.querySelector("#wrapper");
```

document.querySelectorAll()

L'esempio restituisce tutti i tag input di tipo **`text`** (non fattibile con getElementsByTagsName)

```
let txts = document.querySelectorAll("input[type=text]")
for (let txt of txts)  txt.style.backgroundColor="red"
```

Invece di partire da document, si può eseguire la ricerca all'interno di un **`parentNode`**:

```
let _wrapper = document.querySelector("#wrapper");
let txts = _wrapper.querySelectorAll("input[type=text]")
let opts = _wrapper.querySelectorAll("input[type=radio]:checked")
```

Il metodo.contains()

`if (_div1.contains(_div2))`

verifica se _div2 è un **`discendente`** di _div1, cioè se _div2 coincide con _div1 oppure è un suo **figlio** o **nipote**.

Gestione dinamica degli eventi

Le procedure di evento possono essere associate staticamente all'interno del file html oppure dinamicamente tramite codice javascript, nel qual caso sono disponibili 2 diverse sintassi quasi equivalenti.

1. Assegnazione diretta della Property **onclick** analogo all'assegnazione inline nel file HTML

```
_div.onclick = esegui
_div.onclick = function(){ let n = random();  esegui(n) }
```

esegui è un puntatore a funzione che deve tassativamente essere scritto **SENZA parametri** e **SENZA parentesi tonde**. Le parentesi tonde provocherebbero il richiamo immediato della funzione senza eseguire nessuna associazione di evento.
All'evento onclick verrebbe in pratica assegnato il risultato ritornato da esegui()

- **onclick** deve essere scritto tutto in minuscolo
- La **forma anonima** ha il vantaggio che consente di passare dei parametri ad esegui()
- `_div.onclick=`**null**
  rilascia il gestore di evento che, in corrispondenza dei click successivi, non verrà più eseguito.
  Nel caso dei **controlli** per disabilitare la risposta agli eventi è sufficiente impostare **disabled=true**

2. Utilizzo del metodo **addEventListener** (preferibile) che si aspetta due parametri:

- Il nome dell'evento scritto **senza il prefisso on**
- un puntatore a funzione **priva di parametri**. La funzione anche può essere scritta in forma anonima o come named function, **ma sempre senza parametri**

```
_div.addEventListener("click",  esegui )
_div.addEventListener("click",  function(){ esegui(n1, n2) });

_div.addEventListener("click",  function myFunction(){  });
```

Il nome myFunction sarà però visibile soltanto all'interno della funzione stessa !!

removeEventListener rilascia il gestore di evento che, nei click successivi, non verrà più eseguito.

```
_div.removeEventListener("click", esegui)
```

Il secondo parametro (puntatore alla funzione da rimuovere) è obbligatorio e NON può essere omesso

Differenze

Fra le due sintassi ci sono sostanzialmente due piccole differenze:

1) **La prima sintassi consente una unica associazione di evento.**
   Una eventuale seconda assegnazione sovrascrive l'associazione precedente. **Viceversa tramite addEventListener si possono eseguire più associazioni di evento a funzioni diverse.** Notare che se si eseguono più associazioni ad una stessa named function**, l'associazione viene memorizzata una volta sola, per cui la procedura di evento verrà eseguita una sola volta**. Viceversa ciascuna funziona anonima rappresenta un oggetto diverso, per cui se si definiscono più handler di evento relativi a funzioni anonime differenti, questi handler, verranno sempre eseguiti tutti. Stessa cosa (ovviamente) se, tramite un ciclo, si definisce più volte l'associazione ad una funzione anonima.
   Per contro nel caso di addEventListener non è possibile rimuovere i listener definiti tramite **funzione anonima**. **Per poter rimuovere un listener di evento occorre necessariamente usare una named function.**
2) **addEventListener** dispone di un terzo parametro `useCapture` che per default è ***false*** e significa che l'evento viene eseguito durante la bubbling phase (dal più interno al più esterno). Con ***true*** gli eventi vengono invece eseguiti in capturing phase, cioè in ordine inverso rispetto al caso precedente (dal più esterno al più interno).

Note

- Il **(click)** di angular NON è un evento inline come può sembrare, ma viene tradotto con un addEventListener
- **div.style.pointerEvents="none"** rimuove tutti gli eventi relativi al mouse (hover, click, etc)

event

A tutte le procedure di evento dichiarate dinamicamente viene **automaticamente iniettato il parametro event**, che però per poter essere utilizzato dovrà essere scritto esplicitamente come parametro della funzione:

```
_div.addEventListener("click", esegui)
function esegui(event) { event.target.textContent=n }
// oppure in forma anonima :
_div.addEventListener("click", function(event){event.target.textContent=n})
```

this

Tutte le procedure di evento dichiarate dinamicamente diventano metodi di evento dell'oggetto che ha eseguito l'associazione, per cui al loro interno, **senza scrivere NULLA** , **diventa possibile utilizzare l'oggetto `this`** che rappresenta un puntatore all'oggetto che ha scatenato l'evento (esattamente come **event.target**).

```
_div.onclick = esegui
_div.onclick = function(){let n=random(); this.textContent=n}
_div.addEventListener("click", esegui)
_div.addEventListener("click", function(){this.textContent=n});
function esegui() { this.textContent=n }
```

event e this nell'html

Nel caso invece dell'utilizzo di onclick nell'html, sia **event** che **this** devono essere passati esplicitamente come parametro. Tra l'altro passando **event**, **this** diventa inutile.

```
<input type="button" onclick="visualizza(event)">
```

event e this nelle sottofunzioni

Prestare MOLTA attenzione al fatto che, se la funzione di evento richiama a sua volta un'altra sottofunzione, all'interno della sottofunzione l'associazione di **event** e **`this` NON è più valida.** Infatti la sottofunzione non può sapere chi è che l'ha richiamata e quindi a che cosa si riferisce il **`this`**, Nota: Tecnicamente all'interno delle sottofunzioni, **`this`** NON rappresenta l'oggetto che ha scatenato l'evento, ma un generico "spazio" delle funzioni.

In caso di necessità occorre passare **event** e/o **this** in modo manuale ed esplicito:

```
_div.addEventListener("click",  function(){ visualizza(this) })
function visualizza (_this)  {
     alert (_this.textContent); }
```

Parametri facoltativi nelle funzioni di evento

Le funzioni di evento ***non*** possono ricevere parametri (a parte `event`). In fase di definizione è però comunque possibile definire ed inizializzare dei parametri facoltativi (cioè con assegnato un valore di default) che potranno essere utilizzati in caso di chiamate esplicite al fuori dell'associazione di evento

```
function myEventFunction(flag=true){}
```

Accesso agli attributi HTML

E' possibile accedere agli attributi HTML di un elemento in due modi, che però **NON** sono equivalenti:

1. I metodi **`getAttribute( )`** **`setAttribute( )`** e **`removeAttribute( ).`** accedono all'attributo HTML vero e proprio scritto all'interno del file html
```
_img.setAttribute ("id", "wrapper");
_img.setAttribute ("class", " class1 class2");
```
2. **accesso diretto** tramite il **`puntino`** che accede alla proprietà del corrispondente DOM javascript.
```
let _img = document.getElementById("myImg");
_img.id="wrapper";
_img.src="./img/img1.jpg";
```

Default state HTML e Current State del DOM JavaScript

- Gli attributi definiti staticamente all'interno del file HTML rappresentano il cosiddetto ***defaultState*** di un tag, che rimane immutato anche se l'utente a run time modifica il contenuto del tag attraverso l'interfaccia grafica oppure tramite javascript.
- I valori che l'utente inserisce attraverso l'interfaccia grafica oppure attraverso js vengono invece scritti all'interno di una **corrispondente proprietà del DOM JavaScript** che rappresenta il cosiddetto ***currentState***

All'avvio quando il browser crea il DOM della pagina html, carica all'interno del currentState tutti i valori statici del defaultState, per cui inizialmente le due cose coincidono. Ma poi quando l'utente applica una modifica nell'interfaccia grafica oppure nel codice javascript, questa modifica viene salvata soltanto nel currentState, cioè nella corrispondente proprietà del DOM javascript, e quindi le due cose divergono.

getAttribute e setAttribute consentono di leggere/scrivere il ***defaultState*** del controllo. Se il valore del campo viene modificato attraverso l'interfaccia grafica oppure da codice, getAttribute() non se ne accorge.
Allo stesso modo se si usa setAttribute() per modificare il defaultState del campo **dopo** che l'utente ne ha modificato il contenuto tramite l'interfaccia grafica, la modifica NON viene visualizzata sulla pagina.

L'accesso diretto tramite puntino viceversa accede in lettura / scrittura al ***currentState*** dell'attributo, cioè alla corrispondente proprietà del DOM javascript.
Indispensabile nel caso dei **controlli** che possono essere modificati dinamicamente dall'interfaccia grafica, cioè:

- l'attributo **value** dei textBox sia singoli che multiline
- l'attributo **checked** di checkbox e radiobutton
- l'attributo **selectedIndex** di un tag select che non dispone di un *defaultState* html e dunque è accessibile soltanto tramite accesso diretto con il puntino (così come anche il **value** riassuntivo di un ListBox).
- La stessa cosa vale per il **value** di una textArea, che non dispone di un *defaultState* html.

*Notare che se l'utente non apporta variazioni all'interfaccia grafica setAttribute() può essere utilizzato anche per i controlli. Però, ad es nel caso dei **radio button**, in corrispondenze del setAttribute("checked"), occorre eseguire manualmente un removeAttribute("checked") sugli altri radio button, che altrimenti rimangono selezionati*

Le proprietà del DOM javascript possono anche essere accedute con le parentesi quadre :
`let id = div["id"]` equivalente a ***let id = div.id***

Attributi booleani con getAttribute() / setAttribute()

In HTML gli attributi booleani non accettano valori. Volendo assegnare un valore l'indicazione di HTML5 è quello di ripetere **eventualmente** il nome dell'attributo medesimo: **`<button disabled="disabled">`**
L'unico modo per abilitare il pulsante è quello di "rimuovere" l'attributo disabled.

```
btn.setAttribute("disabled")  // oppure
btn.setAttribute("disabled", "disabled")
btn.removeAttribute("disabled")
```

Attributi booleani nel Current State javascript

Le proprietà del DOM sono tipizzate, per cui il currentState vede un booleano vero e proprio :

```
btn.disabled = true / false
```

Attributi personalizzati

Gli attributi personalizzati possono essere creati sia in modo statico con setAttribute sia in modo dinamico con il punto

Gli attributi statici creati mediante **setAttribute()** :

- **accettano come valore SOLO stringhe.** Un eventuale JSON può comunque essere serializzato
- ✓ **vengono visualizzati dagli inspector**
- ✓ **possono essere utilizzati con querySelector** `.querySelectorAll("input[citta=fossano]")`

Gli attributi dinamici creati mediante il **puntino**

- ✓ **accettano come valore numeri interi e oggetti, dunque preferibili.**
- **non vengono visualizzati dagli inspector**
- **non possono essere utilizzati all'interno di querySelector**

Notare che anche gli attributi 'ufficiali', come ad es **value**, vengono visti dall'inspector SOLO se definiti all'interno del file HTML oppure da js tramite setAttribute(), ma non vengono visti se definiti mediante il puntino.

Data-attributes

Anteponendo ad un attributo dinamico personalizzato l'attributo **dataset**, viene creato all'interno del DOM javascript un data-attribute dinamico che però sarà visibile nell'inspector ed utilizzabile all'interno di querySelector

```
div.dataset.userName = 'pippo'
console.log(div.dataset.userName)
```

Notare la differenza tra le seguenti righe in cui la 1° definisce un semplice campo js non utilizzabile in querySelector

```
div.userName = 'pippo'
div.dataset.userName = 'pippo'
```

Visualizzazione nell'inspector ed utilizzo con querySelector

I data-attribute definiti mediante **.dataset** vengono automaticamente replicati in un corrispondente **attributo html statico** visibile nell'inspector ed utilizzabile all'interno di querySelector. Questi attributi saranno visibili ed utilizzabili con un prefisso **data-** Se il data-attribute presenta una lettera maiuscola, a livello html questa viene convertita in minuscolo con l'aggiunta di un ulteriore trattino.

Ad esempio il precedente attributo **userName** verrebbe tradotto con **data-user-name**

```
let div = document.querySelector("div[data-user-name='pippo']")
```

Gli apici intorno al value diventano obbligatori SOLO nel caso di spazi o caratteri speciali

E' anche disponibile la seguente sintassi in cui NON viene specificato il 'value'

```
let div = document.querySelectorAll("div[data-user-name]")
```

restituisce tutti gli elementi che dispongono dell'attributo userName

I **data-attributes** possono normalmente essere inseriti nell'HTML e letti / scritti con getAttribute / setAttribute :

```
<div data-student-name='pippo'> student Info </div>
div.setAttribute('data-user-name', 'pippo')
let name = div.getAttribute('data-user-name')
```

L'unico limite di dataset è che non accetta oggetti che, nel caso, **devono essere serializzati** (altrimenti vengono serializzati come [Object, Object]

Accesso e gestione delle proprietà CSS

Anche nel caso delle proprietà CSS c'è una distinzione fra le **proprietà di stile impostati staticamente** tramite HTML/CSS e le **proprietà di stile impostate dinamicamente** tramite javaScript.
Quando tramite javaScript si assegna un nuovo valore ad una proprietà di stile, questo valore non sovrascrive completamente il valore statico impostato tramite HTML/CSS, il quale, pur non essendo più visibile, rimane memorizzato come valore statico della proprietà.

- Nel momento in cui viene assegnato un nuovo valore tramite javascript, questo valore "maschera" il valore statico, e qualunque funzione di lettura (compresa getComputedStyle) restituisce sempre il valore dinamico
- Nel momento in cui il valore assegnato tramite javascript viene rimosso (tramite assegnazione di **stringa vuota** oppure **none**), automaticamente viene riassegnato all'elemento il valore statico memorizzato all'interno del file HTML/CSS che non può in alcun modo essere rimosso / modificato

Sintassi di accesso:

1. In modo diretto tramite la proprietà **.style**: `_div.style.backgroundColor = "blue";`
   Questa sintassi rappresenta il modo migliore per modificare singolarmente i vari attributi di stile. L'eventuale trattino nel nome dell'attributo (es `background-color`) deve essere sostituito dalla notazione camelCasing.
   In lettura però restituisce *soltanto* le proprietà impostate dinamicamente tramite java script,
   a differenza degli attributi html in cui, in assenza dell'attributo dinamico, viene restituito l'attributo statico.

   `style` è un object che può accedere ai vari campi sia tramite **puntino** sia tramite **parentesi quadre**
   E' chiaramente possibile utilizzare gli attributi composti impostando più valori in una sola riga:
   `_div.style.border = "2px solid black";`

2. I metodi **`.getAttribute("style") / .setAttribute("style")`** sono simili ai precedenti, però leggono / scrivono l'intero attributo **style** in un sol colpo, per cui hanno senso SOLTANTO quando effettivamente interessa leggere / scrivere TUTTE le proprietà di stile insieme.

   `.getAttribute`("style") restituisce tutte le proprietà di stile dell'oggetto corrente, sotto forma di ***stringa serializzata***, però **SOLTANTO** quelle impostate dinamicamente tramite javaScript e **NON** quelle impostate staticamente in HTML/CSS, e nemmeno quelle impostate tramite setAttribute ("class", "className")

   `.setAttribute("style",` `"font-size:100px; font-style:italic; color:#ff0000");`
   consente di definire contemporaneamente più CSS property (con un'unica stringa CSS), però **sovrascrive** tutte le proprietà di stile eventualmente già impostate dinamicamente tramite javascript.

   `.removeAttribute`("style") rimuove tutte le proprietà di stile impostate tramite java script

3. **`_div.style.cssText +=`** `"font-size:100px; font-style:italic; color:#ff0000";`
   Utilizzabile sia in lettura che in scrittura, va bene nel caso in cui si desideri impostare più proprietà insieme.
   In lettura esegue in pratica una serializzazione della proprietà style esattamente come getAttribute("style")
   In scrittura **consente il concatenamento** per cui diventa possibile aggiungere nuove proprietà CSS all'elemento corrente senza rimuovere quelle precedentemente impostate.
   Anche questa proprietà 'vede' soltanto le proprietà di stile impostate dinamicamente in javaScript.

Accesso in lettura agli attributi di stile impostati tramite CSS statico

La funzione **`getComputedStyle(refPointer)`** consente di accedere in **lettura** a TUTTE le proprietà CSS dell'elemento corrente, sia quelle create dinamicamente in JavaScript, sia quelle create staticamente in HTML.

```
if(getComputedStyle(_div).visibility=="hidden")
            _div.style.visibility="visible";
```

- **Se esiste un valor dinamico restituisce il valore dinamico.**
- **Se non esiste un valore dinamico restituisce i valori statici memorizzati nel file CSS.**

Note Operative

1) **`getComputedStyle`** restituisce i **`colori`** sempre in formato RGB e dunque i confronti vanno fatti sull'RGB

```
if(getComputedStyle(div).backgroundColor != "rgb(234, 234, 234)")
```

viceversa il semplice **`.style`** restituisce il valore dinamico così come è stato impostato.

2) **`backgroundImage`** restituisce una url sempre scritta tra apici doppi interni. Es **`url("img/img1.gif")`**
Per cui i confronti vatti sempre utilizzando gli apici doppi interni

```
this.style.backgroundImage = `url("img/img1.gif")`
```

Le stringhe su righe multiple

Il **backtick** consente di andare a capo all'interno di una stringa.

Prima dell'introduzione del backtick, per poter scrivere una stringa su righe multiple occorreva terminare ogni riga con il carattere backslash \ che però doveva necessariamente essere l'ultimo carattere della riga, senza spazi successivi:

```
let s = "salve \
         mondo";
```

Il passaggio dei parametri in javascript

I numeri e le variabili scalari in genere vengono **copiati**, **passati** e **confrontati** per valore e, come in java, NON è possibile passarli per riferimento. Eventualmente la procedura può ritornare il valore modificato.

Vettori, Matrici e Oggetti sono sempre **copiati**, **passati** e **confrontati** per riferimento e, come in java, NON è possibile passarli per valore.
Il confronto fra due oggetti identici restituisce false a meno che i puntatori non stiano puntando allo stesso oggetto.

Le **stringhe** vengono sempre **copiate** e **passate** per riferimento come gli obiect.
Le stringhe dichiarate con il **new** vengono **confrontate** per riferimento, come tutti gli Obect.
Le stringhe dichiarate senza il **new** vengono **confrontate** per valore.
Se uno o entrambi i valori è una semplice stringa (senza il new), allora il confronto viene eseguito per valore.

Accesso diretto ai controlli di una form

Per accedere direttamente ai controlli di una form si può utilizzare il **name** della form seguito dal **name** del controllo

```
let nomeUtente = document.form1.txtUser.value
```

Nota: Utilizzando questa sintassi, nel caso di un vettore di radio buttons aventi tutti **`name="opt"`**, l'istruzione

```
form1.opt.value
```

restituisce il **`value`** della voce attualmente selezionata. Però vale solo in questo caso.
Utilizzando invece **`document.getElementsByName(opt)`** non funziona più.

Gli oggetti localStorage e sessionStorage

Per ragioni di sicurezza i browser impediscono a javascript l'accesso al file system della macchina locale.
Cioè non è consentita una istruzione del tipo

```
let content = readFile("c:\cartella1\file2.txt");
alert (content);
```

Poiché talvolta una applicazione ha necessità di salvare delle informazioni sulla macchina locale, è stato aggiunto alle librerie java script un apposito oggetto denominato **localStorage** che consente ad ogni applicazione di salvare fino a 10 MB di dati sul **HD** del PC locale, all'interno di un'area gestita dal browser e non direttamente visibile all'utente., ma comunque visibile dell'inspector.

Le variabili salvate all'interno del LocalStorage sono suddivise per **dominio**.
Cioè quando siamo connessi ad un certo dominio sono visibili ed accessibili soltanto le variabili relative a quel dominio (con un max di 10MB per dominio)

L'oggetto **sessionStorage** salva invece i dati soltanto nella memoria del browser, per cui vengono persi nel momento in cui il browser viene chiuso e risultano disponibili soltanto per la pagina che li ha creati.

All'interno degli oggetti localStorage e sessionStorage si possono salvare variabili (dette **item**) in formato chiave-valore. Le variabili vengono salvate nell'area relativa al dominio corrente.

```
localStorage.setItem("key1", "value1");
let value1 = localStorage.getItem("key1");
localStorage.removeItem("key1");
localStorage.clear();  // ripulisce l'intera localStorage relativo al dominio
localStorage.length;   // restituisce il numero di chiavi memorizzate
```

Note

1) Il **value** può essere solo di tipo **string**. Eventuali numeri, booleani, vettori, json vengono automaticamente serializzati. Il problema è che i json vengono serializzati come [object Object]
2) In **Firefox** l'uso del local Storage funziona **SOLTANTO** con il protocollo **http**, in cui gli item sono salvati per dominio. Con il protocollo **file** gli item sono salvati per pagina, quindi una pagina non vede gli item delle altre pagine.
3) Per vedere se il browser supporta la localStorage si può utilizzare una delle seguenti condizioni:
   ```
   if (typeof(localStorage) != "undefined") {
   if('localStorage' in window && window['localStorage'] !== null) {
   ```
4) Attenzione al fatto che, avviando l'applicazione con webstorm o visual studio code, questi creano un server locale di accesso alle pagine. Per cui se questi server creano un file in localStorage, aprendo la stessa pagina tramite file system con un doppio click, questa NON vedrà il file all'interno del local Storage e viceversa.

La creazione dinamica degli elementi

In Java Script è possibile creare tag dinamicamente ed aggiungerli all'interno di altri tag utilizzando un tipico modello ad albero. I metodi da utilizzare sono:

- `let ref = document.createElement("tagName") // creo un nuovo tag`
- `parent.appendChild(ref)  // lo appendo ad un tag esistente`

Esempio di creazione di una nuove righe e celle all'interno di una tabella :

```
let tabella = document.getElementById("mainTable");
for (let i=0; i<DIM; i++){
    let tr = document.createElement("tr")
    tabella.appendChild(tr)
    for (let j=0; j<DIM; j++){
        td = document.createElement("td")
        tr.appendChild(td)
        let img = document.createElement("img")
        td.appendChild(img);
        img.id=`img-${i}-${j}`
        img.addEventListener("click", cambiaImmagine)
        let span = document.createElement("span")
        span.innerHTML+="immagine"+j;
        td.appendChild(span);
    }
}
let btn = document.createElement("input");
btn.type = "button"
btn.value = "Elabora"
btn.addEventListener("click", elabora);
```

Per accedere ad un elemento figlio del nodo corrente si può usare .**childNodes[i]** a base 0

```
let div = wrapper.childNodes[2]  // terzo figlio
```

Per vedere se l'elemento corrente ha figli si può usare .**hasChildNodes**

```
if(wrapper.hasChildNodes) .....
```

Note

1) Esiste anche un metodo **.append()** simile ad **appendChild()** ma che consente di appendere più elementi contemporaneamente, separati da virgola. Inoltre, rispetto ad appendChild(), append() accetta come parametro anche stringhe html ma non ritorna il puntatore all'elemento parent comodo per le chiamate in cascata.
2) Nel metodo **parent.appendChild(elem)**, se l'elemento ricevuto come parametro è già appeso al DOM, viene automaticamente 'tagliato' ed appeso nella nuova posizione.
3) Nel caso di **createElement("input")** si può specificare un secondo parametro che indica il tipo di input che si intende creare : **createElement("input", "text")**
4) Per alcuni elementi, in alternativa a `document.createElement()`, è disponibile anche l'operatore **new**
   let opt = new Option(text, value) // Opzione da aggiungere ad un select
5) `document.createTextNode("testo")` crea un nodo testuale che può essere appeso a qualsiasi tag. Analogo all'utilizzo della property **.textContent**

Nota sul concatenamento di stringhe all'interno di `innerHTML`

Supponiamo di avere all'interno della pagina html un tag DIV con **`id=wrapper`** al quale andiamo ad appendere tramite java script altri tag creati dinamicamente. Se ad un certo punto si utilizza una istruzione dei tipo

```
_wrapper.innerHTML += "<br>"     // oppure += qualunque altra cosa
```

il **concatenamento** all'interno della proprietà **`innerHTML`** forza nel browser una **rigenerazione dell'intero contenuto del tag wrapper**, cioè serializza l'oggetto, aggiunge il nuovo contenuto e poi parsifica di nuovo.

Per cui eventuali puntatori javascript agli oggetti creati dinamicamente all'interno di wrapper vengono tutti persi. Insieme ai puntatori vengono persi proprietà e metodi assegnati dinamicamente al puntatore, cioè

- le proprietà HTML assegnate dinamicamente (sia quelle appartenenti al DOM sia quella personalizzate)
- i metodi di evento assegnati dinamicamente tramite addEventListener

questo perché il puntatore usato in fase di creazione per definire l'evento NON punta più al nuovo oggetto ridisegnato all'interno di wrapper.

Per cui, quando si lavora con gli oggetti, occorre abbandonare definitivamente il concatenamento di stringhe. Può essere utilizzata in assegnazione ma NON in concatenamento !! Usare SEMPRE createElement()

Tecniche per la creazione di una matrice di oggetti interni ad un contenitore

Supponiamo di avere un tag statico `<div id="wrapper">` e di volerlo riempire con 10x10 = 100 tag DIV da creare dinamicamente ed appendere a wrapper. Si possono utilizzare diverse soluzioni:

1) Utilizzare una tabella come nel caso precedente. A questo punto il wrapper potrebbe essere di tipo <table> al quale si aggiungono poi i <TR> e infine i <TD>. Soluzione perfetta per visualizzare tabelle di dati ma poco adatta ai giochi (spaziature indesiderate fra celle e anche fra righe)

2) Utilizzare un contenitore di tipo <DIV> e impostare sugli elementi interni **`float:left`** oppure ancora meglio **`display:inline-block`**. Gli elementi interni vengono disposti uno a fianco dell'altro fino al raggiungimento del margine destro del contenitore, in corrispondenza del quale vanno automaticamente a capo. A tale scopo diventa fondamentale la larghezza del contenitore che deve essere esattamente 10 volte la larghezza degli elementi interni. A volte però (gioco della roulette) il contenitore necessita di un maggior spazio vuoto a destra. In tal caso si potrebbe agire sul padding-right del contenitore

3) Utilizzare un contenitore di tipo <DIV> e con **`position:relative`** ed assegnare **`position:absolute`** agli elementi interni. In questo caso ci svincoliamo dalla larghezza del contenitore

4) Si possono lasciare gli elementi interni così come sono (display:block) e costruire il corpo per righe creando prima un tag <DIV> con **`display:flex`**, e poi aggiungendo al suo interno i 10 tag <DIV> che andranno a costituire la riga, simulando una tabella. Anche in questo caso ci svincoliamo dalle dimensioni del contenitore.

Accesso alle righe di una Tabella

Il puntatore a tabella presenta una interessante proprietà **`rows`** che rappresenta un vettore enumerativo contenente i puntatori alle varie righe che costituiscono la tabella. A sua volta la riga contiene una collezione di **`cells`**

```
let _table = document.getElementById("table")
_table.rows.count = numero complessivo di righe (compresi header e footer)
_table.tBodies[0].rows.count = numero di righe del tBody
_table.rows[i].cells  = vettore delle celle della riga corrente
_table.rows[i].cells[j]  = cella j-esima
```

Notare che rowSpan e colSpan si scrivono in camel-casing:

```
td.rowSpan = 3;
```

```
  if(_table.rows.length > 0) {
     let tr = _table.rows[i];
     if(tr.cells.length > 0)
          let cella = tr.cells[j]
```

Gestione dell'errore nel caricamento di una immagine

Quando si setta l'attributo **src** di una immagine, automaticamente viene richiesta l'immagine al server ed src punterà all'immagine in memoria. Se l'immagine indicata nel path non esiste, viene generato un evento **error** che può essere gestito con le solite due sintassi javascript:

```
_img.addEventListener("error", function() {
    this.src = "default.jpg"  })

_img.onerror = function () {
    this.src = "default.jpg"  }
```

Attenzione che i nomi dei file contenenti le immagini **NON devono contenere spazi** (che è un carattere non ammesso all'interno di una URL). Se il nome del file contiene uno spazio, questo spazio da codice potrà essere sostituito con %20 (codifica esadecimale dello spazio), che però in talune applicazioni crea problemi.

Drag and Drop

```
<div id="dragItem" draggable="true">Drag me</div>
<div id="dropZone">Drop here</div>

// drag-start
dragItem.addEventListener("dragstart", function(e) {
    // dataTransfer è l'oggetto che si porta dietro l'evento drag&drop
    // Le informazioni possono essere recuperate dentro il drop
    // non si più passare un riferimento ad object
    e.dataTransfer.setData("index", 0);
    dragItem.style.opacity = "0.5";
});

// drag-end
dragItem.addEventListener("dragend", function() {
     dragItem.style.opacity = "1";
});

// dropzone drag-over
dropZone.addEventListener("dragover", function(e) {
    // Per impostazione predefinita, un elemento non può ricevere un drop
    // quindi bisogna sbloccarlo
    e.preventDefault();
    dropZone.style.background = "#eee";
});

// dropzone drop
dropZone.addEventListener("drop", function(e){
   // e.preventDefault(); non serve
    if (dropZone.children.length==0){
        dropZone.textContent = ""
        dropZone.style.justifyContent = "flex-start"
    }
    dropzone.style.background = "";
    const index = e.dataTransfer.getData("index");
    dropZone.appendChild(dragItem);       // oppure
    dropZone.appendChild(drags[index]);   // oppure
    const clone = drags[index].cloneNode(true)   // .clone è jQuery
    dropZone.appendChild(clone);
});
```

L'oggetto window: proprietà, metodi ed eventi

name E' il nome assegnato ad una finestra aperta da codice.

closed Dalla finestra attuale è possibile creare una nuova finestra mediante il metodo open ricevendo un puntatore alla finestra. Con questo puntatore è possibile verificare lo stato *closed* della nuova finestra

status Contenuto della barra di stato inferiore. Passando il mouse su un collegamento ipertestuale, il browser visualizza automaticamente nella barra di stato inferiore l'URL completo del link.
Java Script può modificare questo msg mediante l'evento **onMouseOver.** Però se si vuole sostituire l'azione di default con una azione utente, occorre restituire al gestore onMouseOver il valore **true**. Altrimenti l'azione di default maschera l'azione utente.
<a href="pagina3.htm" onMouseOver="window.status='Caratteristiche Tecniche'; **return true**">

defaultStatus Messaggio iniziale visualizzato nella barra di stato dopo il caricamento di una nuova pagina

height altezza della finestra

setInterval

Sia `setTimeout` che `setInterval` sono asincrone, cioè avviano la procedura all'interno di un '*thread*' separato, per cui eventuali istruzioni successive a setInterval / setTimeout vengono eseguite subito dopo.

setInterval( ) richiede due parametri:

- Un puntatore a funzione
- Un tempo espresso in millisecondi

Esempio:

```
let timerID=setInterval(visualizza, 1000);
```

La procedura **`visualizza`** verrà richiamata ciclicamente a intervalli regolari di 1000 msec, cioè ogni secondo, in modo analogo all'oggetto timer di C#.

setInterval() restituisce un ID che può essere utilizzato per arrestare il temporizzatore :

```
if(timerID) clearInterval(timerID)
```

Per riavviarlo occorre riscrivere l'intera istruzione setInterval()

Esempio di visualizzazione dell'ora corrente

```
function visualizzaOraCorrente() {
     let d = new Date();
     _div.innerHTML = d.toLocaleTimeString();
}
```

In alternativa si può incrementare una variabile globale `seconds` ad intervalli di 1000 msec.

```
seconds++;
if(seconds%60==0) {minutes++;  seconds=0; }
```

setTimeout

setTimeout( ) è analogo a setInterval() ma la funzione indicata **viene eseguita una sola volta**

```
let timerID =setTimeout(visualizza, 1000)
```

`visualizza` viene avviata dopo 1 sec dal richiamo di setTimeout() e **viene eseguita una sola volta** a meno che, al termine della procedura medesima, venga di nuovo richiamato setTimeout() che la fa ripartire un'altra volta.

Esattamente come setInterval() restituisce un **ID diverso da 0** che consente di disabilitare il timer prima dello scadere del tempo indicato:

```
if(timerID) clearTimeout(timerID)
```

Ritardo della alert()

In condizioni normali la alert() viene richiamata prima del rendering grafico della pagina, per cui non si vedono gli ultimi aggiornamenti. Il problema può essere risolto ritardando la visualizzazione della alert con un setTimeout:

```
setTimeout(function(){alert("hai vinto")}, 50)
```

Passaggio di un parametro

Sia nel caso di **setInterval** che nel caso di **setTimeout** è possibile passare un terzo parametro che rappresenta una variabile che verrà automaticamente iniettata alla funzione richiamata da setInterval / setTimeout

```
let timerID=setInterval(visualizza, 1000, myVar);
function visulizza(myVar){
   console.log(myVar)    // Il valore di myVar viene visualizzato ogni sec
}
```

window.open

open("file.htm", ["target"], ["Opzioni separate da virgola"])

Il **primo parametro** indica il file da caricare. Se si specifica "", verrà aperta una nuova scheda vuota.

Il **secondo parametro target** rappresenta la scheda di apertura della pagina e può assumere i valori "_blank", "_self", etc. oppure un nome alfanumerico (**TARGET**) nel qual caso il file verrà aperto in una nuova scheda a cui verrà assegnato il target indicato.

```
<a href="#" onClick='window.open("pag2.htm", "Finestra2");'> apri </a>
<a href='TerzaPagina.html'  target="Finestra2"> vai</a>
```

TerzaPagina.html verrà aperta all'interno della scheda Finestra2 creata da window.open()

Il terzo parametro consente di aprire la pagina **in una nuova finestra** e consente di esprimere le caratteristiche della nuova finestra. In tal caso come secondo parametro si può impostare stringa vuota oppure un target identificativo. I vari parametri devono essere scritti **senza spaziature.**
window.open('pagina2.htm', '', 'resizable=no, width=300, height=300, left=320, top=230, fullsceen=no, menubar, toolbar=no, scrollbars=yes, status=no');
Il valore yes può anche essere omesso scrivendo soltanto il nome dell'opzione.

Opzioni terzo parametro:

Nome	Valore	Spiegazione
width height	Numerico pixel	Larghezza – Altezza della finestra
left	Numerico pixel	Distanza dalla sinistra del monitor
top	Numerico pixel	Distanza dal lato superiore del monitor
fullscreen	yes / no	Apertura a tutto schermo
menubar	yes / no	Presenza del menù
toolbar	yes / no	Presenza della toolbar
scrollbars	yes / no	Presenza delle scroll bar
status	yes / no	Presenza della status bar in basso
location	***yes / no***	***Presenza della barra degli indirizzi***
resizable	***yes / no***	***Ridimensionabile***

location e resizable sembrano deprecati. Al loro posto si può usare il widget dialog di jQueryUi che è basato non su windows.open ma sulle inline dialogs, cioè la visualizzazione di un tag DIV in primo piano con oscuramento della parte sottostante.

Il metodo open viene spesso sfruttato per aprire banner pubblicitari

```
<a href="pagina2.htm" onClick='window.open("banner.htm", " NuovaFinestra ");'> Vai a pagina2 </a>
<body onLoad='window.open("banner.htm", " NuovaFinestra ");'>     oppure body onUnload
```

Gestione del riferimento alla finestra aperta da open

Il metodo open restituisce un puntatore alla nuova finestra appena aperta. Esempio:

```
let _div = window.open("pagina2.htm", "NuovaFinestra");
```
_div .close() // Chiude la nuova finestra
_div = null // Dopo la chiusura di una finestra è bene rilasciare il puntatore

La proprietà opener

Ogni finestra (window) ha una interessante proprietà **opener** che è un puntatore alla finestra o frame che ha generato la sottofinestra mediante window.open(). Per la finestra principale opener = null. Esempio:

```
<input type="text" onChange = "opener.document.getElementById().value="x">
```

window. beforeunload

```
window.addEventListener('beforeunload', function () {  }
```

Questo evento viene richiamato quando la finestra corrente sta per essere chiusa (a seguito del click sulla X oppure in seguito alla chiusura del browser), *prima* che il DOM venga deistaziato

- **Consente** l'accesso agli elementi del DOM (non ancora deistaziato)
- **Consente** di rilasciare delle risorse allocate dalla pagina (come ad esempio l'esecuzione di una chiamata Ajax per informare il server della chiusura della pagina)
- **Non Consente** di eseguire delle alert() o aprire finestre di dialogo in genere

Altre Proprietà e metodi dell'oggetto document

title E' il titolo della pagina impostato nella head dal tag title
lastModified Data e ora dell'ultima modifica della pagina

Il metodo document.write (s)

Consente di scrivere dinamicamente il contenuto di una **nuova** pagina creata con window.open()
All'interno della stringa s può essere inserito qualunque tag html.
Se il metodo viene eseguito su una pagina già caricata, write troverà il documento chiuso e provvederà a rimuoverlo sostituendolo con un documento vuoto in cui andrà a scrivere il contenuto di s.

Creazione dinamica di un nuovo documento

Dopo aver creato una nuovo finestra vuota tramite window.open()

```
let w=window.open("","_blank");
```

è possibile inserire il contenuto nel modo seguente:

```
w.document.write("<h1 align='center'>Titolo della nuova pagina</h1>");
```

Metodi dell'oggetto document per la scrittura dinamica:

open() Apre un documento in scrittura. Opzionale. Se il documento è chiuso write lo apre automaticamente
write (s) Se usato all'interno di una pagina vuota consente di creare dinamicamente il contenuto della pagina. Il contenuto di s viene scritto alla posizione attuale del cursore. **All'interno della stringa s può essere inserito qualunque tag html** compreso \n Il flusso di output viene però automaticamente chiuso al termine del caricamento della pagina. Dunque se il metodo viene eseguito verso una pagina già caricata, write troverà il documento chiuso e provvederà a rimuoverlo sostituendolo con un documento vuoto in cui andrà a scrivere il contenuto di s.
writeln (s) Come write() con in più il ritorno a capo aggiunto automaticamente al fondo di s
close () Serve per chiudere il flusso al termine delle write. Sebbene il flusso venga chiuso automaticamente al termine del caricamento della pagina, i manuali consigliano di eseguire sempre il close() subito dopo l'ultimo write. Altrimenti potrebbero esserci problemi nel caricamento di immagini e moduli

OGGETTO window.location

Contiene tutte le informazioni sulla URL corrente

Proprietà e Metodi

href URL attuale completa http://indirizzo. **Proprietà predefinita di location, per cui può anche essere omessa**. Modificare la proprietà href dell'oggetto location è il modo più semplice per caricare una nuova pagina mediante uno script: `location.href="pagina3.htm"` oppure anche da HTML:

```
onclick="window.location.href='home.html'"
```

Accetta come parametro anche un'ancora interna alla pagina corrente
`window.location.href='#ancora'`

Notare che l'impostazione della proprietà **`href`** **NON termina l'elaborazione dello script** che prosegue eseguendo eventuali istruzioni successive. Per terminare lo script si può utilizzare:

- `return false`; termina la funzione in corso
- **`window.stop()`**; termina l'intero script

Nota: All'interno di href, come in html, si può specificare un indirizzo di posta elettronica, preceduto da mailto: in tal caso verrà aperto il client di posta predefinito. All'indirizzo di posta possono essere concatenati anche dei parametri riguardanti ad esempio il body da preimpostare.

reload( ) Ricarica l'intero documento (come il tasto Reload del browser)
Ha un parametro facoltativo che ha un valore di default pari a **false** nel qual caso il reload viene fatto dalla cache se possibile). Impostando **true** viene forzato il reload dal server.

replace("URL") Carica una nuova pagina nella finestra corrente. Rispetto alla precedente elimina la pagina attuale dalla cronologia. Facendo INDIETRO l'utente non vedrà più la pagina corrente ma ritornerà alla pagina antecedente. Utile per eliminare dalla cronologia pagine intermedie utilizzate in una certa fase.

protocol protocollo di accesso alla risorsa. Es **http**, **file**, **ftp**.
hostname nome del dominio richiesto
port porta di comunicazione. 80 nel caso di http, stringa vuota nel caso del procollo *file*
host hostname : port

pathname risorsa richiesta
search restituisce la ***`queryString`*** della url comprensiva del ?
hash = "Capitolo2" Consente di navigare verso un nuovo ancoraggio presente nella pagina

OGGETTO window.history

Contiene tutte le informazioni relative alle URL visitate prima e dopo rispetto alla URL attuale.
Consente la navigazione avanti e indietro attraverso la storia della finestra corrente.

Proprietà

length Numero di pagine visitate precedentemente rispetto alla pagina attale
current URL della pagina attualmente caricata
previous URL della pagina precedente nella cronologia
next URL della pagina successiva nella cronologia (ha senso solo se si è usato il pulsante back)

Metodi

back()	Ritorna alla pagina precedente. La pagina **viene ricaricata**, però vengono automaticamente ripassati al server eventuali parametri get e post (esattamente come avviene con il pulsante BACK del browser).
forward()	Va avanti di una pagina (se esiste)
go(-1)	Va avanti / indietro ad una posizione ben definita. go(-2) torna indietro di 2 pagine. La pagina **viene ricaricata** con il passaggio automatico dei parametri get e post, esattamente come avviene per il metodo back() e per il pulsante BACK del browser.

OGGETTO navigator

Al momento dell'apertura del browser, viene allocato un oggetto NAVIGATOR, fratello dell'oggetto WINDOW, contenente tutte le informazioni sul browser che si sta utilizzando. Questo oggetto rimane allocato in *unica* istanza fino alla chiusura del browser.

appName	Nome del browser. Es Microsoft Internet Explorer
appVersion	Versione del browser Es versione 4.0 (compatible; MSIE 5.5; Windows 98)
appCodeName	Nome in codice del browser. Es "Mozilla"
userAgent	E' la stringa di intestazione inviata all'host quando gli si richiede una pagina web. Contiene informazioni sul browser, sul sistema operativo e sulle rispettive versioni

Approfondimenti

L'operatore ===

Confronta non solo il valore ma anche il tipo

```
let a = 1;
let b = "1";
if (a==b)   // true
if (a===b)  // false
```

Utilizzo dell'operatore || sulle stringhe

In java script è possibile eseguire una OR fra due o più stringhe.
Il risultato è pari al contenuto della prima stringa che presenta una valore **diverso da undefined**.

```
let ris  = stringa1 || stringa2 || stringa3
      let a;
      let b = "hi"
      console.log(a||b)    // "hi"
```

Numero arbitrario di parametri

```
visualizza("pippo", "pluto", "minnie");
function visualizza(..args){
     console.log(args)     // ["pippo", "pluto", "minnie"]
}
```

dove args è semplicemente un nome assegnato localmente al parametro.
…args restituisce un vettore enumerativo dei parametri

Notare che questa sintassi può essere adottata in qualunque funzione.
Sta al chiamante passare i parametri corretti. Il chiamato li potrà leggere dentro **`args[index]`**

Parametri opzionali

```
function ricerca(param1, param2 = false) {}
```

Se il chiamante non passa il secondo parametro, **param2** viene automaticamente settato a false

The ternary conditional operator

E' una tecnica disponibile in tutti i linguaggi per compattare al massimo il costrutto if.

Si supponga di dover eseguire le seguenti assegnazioni:

```
if(ok)  msg = 'yes';
else    msg = 'no';
```

Questo costrutto può essere riscritto in modo molto più conciso nel modo seguente:

```
let msg = ok ? 'yes' : 'no';
```

Cioè se la variabile **ok** è vera, alla variabile **msg** viene assegnato yes, altrimenti viene assegnato no.

Note:

1) Attenzione al fatto che, dopo il ?, occorre necessariamente utilizzare o dei valori diretti (come nell'esempio) oppure delle funzioni che restituiscono un valore. Non è consentito utilizzare delle procedure perché non potrebbero assegnane nessun valore a msg
2) **msg potrebbe anche essere omesso**, nel qual caso il costrutto si limita ad eseguire una delle due funzioni di destra a seconda del valore di ok. Anche in questo caso però a destra non sono ammesse procedure ma sempre soltanto funzioni.

Assegnazione di una booelana tramite condizione diretta

```
let ok = (a > 0)
```

Se `(a > 0)` allora ad ok viene assegnato il valore `true`, altrimenti viene assegnato il valore `false`

try and catch

```
try {
      alert("Welcome guest!");
}
catch(err) {
      document.getElementById("demo").innerHTML = err.message;
}
```

Il player audio

Il player audio, inserito tramite il tag <audio> deve essere inizializzato assegnando all'attributo **`.src`** il path della canzone, esattamente come il tag img.

- La riproduzione può essere avviata da codice tramite il metodo **`.play()`** oppure cliccando manualmente sul pulsante play.
- In entrambi i casi viene generato un evento **onplay** che può essere sfruttato ad esempio per incrementare il numero degli ascolti
- Se il file della canzone non è presente, viene generato un evento **onerror** esattamente come per il tag img.

Il tag <svg>

Il tag <svg> consente di visualizzare una immagine vettoriale controllando colori e dimensioni da css/js

```
<svg width="800px" height="800px" viewBox="0 0 1024 1024" class="icon"
version="1.1" xmlns="http://www.w3.org/2000/svg">
   <path>   </path>
</svg>
```

`width` = larghezza dell'immagine
`width` = altezza dell'immagine
impostandole entrambe l'immagine può risultare deformata, a meno di usare
`preserveAspectRatio="xMidYMid meet"`
che significa centrata in orizzontale e verticale e interamente visibile (backgroundSize=contain)
`preserveAspectRatio="xMidYMid slice"`
che significa centrata in orizzontale e verticale coprendo tutto lo spazio (backgroundSize=cover)

`viewBox="min-x min-y width height"`
definisce il sistema di coordinate interno.
`viewBox="0 0 1024 1024"`
significa che il disegno viene fatto in un'area 1024×1024 e poi riscalato automaticamente 800 x 800

`class="icon"`
assegna una classe CSS definita a parte.
Ad esempio `.icon {fill:red;  width:40px;  height:40px;}`

`version` = indica la versione della specifica SVG. 1.1 è la più utilizzata

`xmlns` = Definisce il namespace XML. Dice al browser "Questo è un SVG, non HTML"

`<path>` contiene il disegno vero e proprio

DEFER e ASYNC

L'attributo **defer** indica al browser di caricare lo script "in background", parallelamente al caricamento della pagina, e poi eseguire lo script quando è caricato.

```
<script defer src="index.js></script>
```

- Gli script con defer non bloccano il caricamento la pagina
- Gli script con defer vengono sempre eseguiti SOLTANTO al termine del caricamento del DOM, subito prima prima dell'evento window.onload()
- L'attributo defer viene ignorato se il tag <script> non ha l'attributo src.

In sostanza l'attributo **defer** rappresenta una interessante alternativa all'evento **window.onload()**

ASYNC

L'attributo **async** indica al browser di caricare lo script "in background" (esattamente come defer) ma in questo caso lo script verrà eseguito appena pronto. Il DOM e gli altri script non restano in attesa che questi vengano caricati. Avremo quindi un script completamente indipendente che verrà subito eseguito al termine del caricamento.

- L'evento DOMContentLoaded potrebbe scattare sia prima che dopo rispetto ad uno script async, dipende dalla pesantezza dello script.
- Allo stesso modo gli script async non si aspettano a vicenda. Uno script più piccolo come small.js inserito nella pagina per secondo probabilmente verrà caricato prima di long.js e quindi anche eseguito per primo. Questo ordine viene chiamato "load-first".

Gli script async sono ottimali quando dobbiamo integrare uno script di terze parti indipendente: contatori, ads, e così via, visto che essi non dipendono dai nostri script e i nostri script non devono aspettare il loro caricamento:

Script dinamici

E' anche possibile aggiungere uno script dinamicamente tramite JavaScript:

```
let script = document.createElement('script');
script.src = "myscript.js";
document.body.append(script);
```

Lo script inizia a caricarsi nel momento in cui viene appeso a document. Il caricamento è esattamente come quello degli script "async", cioè:

- Non aspettano nessun altro script e nessuno aspetta loro.
- Lo script che viene caricato per prima viene anche eseguito per primo (ordine "load-first")

E' possibile cambiare l'ordine "load-first" nell'ordine con cui vengono richiamati (come per i normali script) settando l'attributo async a false:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  script.async = false;
  document.body.append(script);
}

loadScript("long.js");
loadScript("small.js");
```

Senza script.async=false gli script verrebbero eseguito secondo l'ordine load-first (small.js probabilmente per primo). Ma con il flag `async = false` l'ordine diventa "come nel documento", per cui long.js verrà eseguito prima di small.js

Accesso diretto agli elementi del DOM

E' possibile accedere direttamente a tutti gli elementi del DOM attraverso il loro ID

```
window["btnIndietro"].disabled=true;
```

Allo stesso modo è possibile accedere anche alle variabili globali:

```
let a = 15;
window["a"]++;
alert(a);       // 16
```

Lettura dei parametri GET

```
function leggiParametriGet(){
    let json = {};
    let parametri = [];
    let s = window.location.search;
    // estraggo dal punto interrogativo in avanti
    s = s.substr(s.indexOf("?") + 1);
    // sostiuisco %20 con " "
    let exp = new RegExp("%20", "g");
    s = s.replace(exp, " ");
    parametri = s.split("&");

    let parametro = [];
    for (let i = 0; i < parametri.length; i++)
    {
      parametro = parametri[i].split("=");
      let key = parametro[0];
      let value = parametro[1];
      // Se il nome del parametro termina con [], compatto i valori in una stringa
      if(key.substr(key.length-6, 6)=="%5B%5D"){
            key=key.substr(0,key.length-6);
            if (!(key in json))
                  json[key] = value;
            else
                  json[key]+=", " + value;
      }
      else
            json[key] = value;
    }
    return json;
```

Un sito di rapido test del codice

www.webtoolkitonline.com